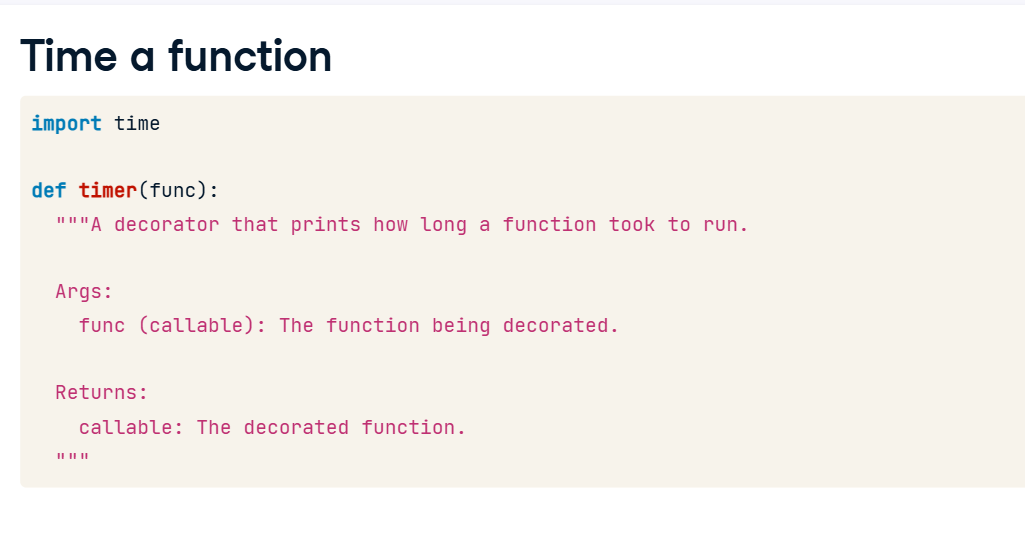
**Real-world examples**

You've learned a lot about how decorators work. This lesson will walk you through some real-world decorators so that you can start to recognize common decorator patterns.

**2. Time a function**

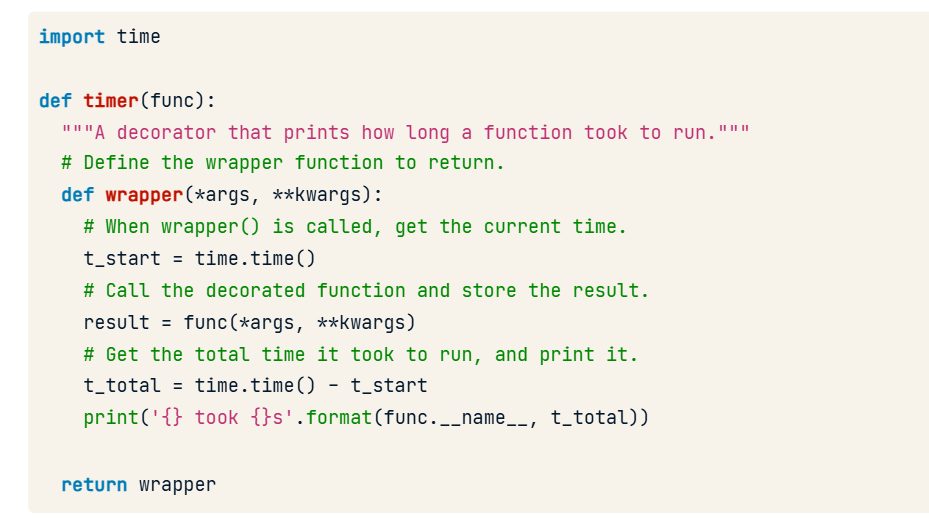
00:10 - 00:38

The timer() decorator runs the decorated function and then prints how long it took for the function to run. I usually wind up adding some version of this to all of my projects because it is a pretty easy way to figure out where your computational bottlenecks are. All decorators have fairly similar-looking docstrings because they all take and return a single function. For brevity, I will only include the description of the function in the docstrings of the examples that follow.



**3. Time a function**

Like most decorators, we'll start off by defining a wrapper() function. This is the function that the decorator will return. wrapper() takes any number of positional and keyword arguments so that it can be used to decorate any function. The first thing the new function will do is record the time that it was called with the time() function. Then wrapper() gets the result of calling the decorated function. We don't return that value yet though. After calling the decorated function, wrapper() checks the time again, and prints a message about how long it took to run the decorated function. Once we've done that, we need to return the value that the decorated function calculated.



**4. Using timer()**

01:20 - 01:40

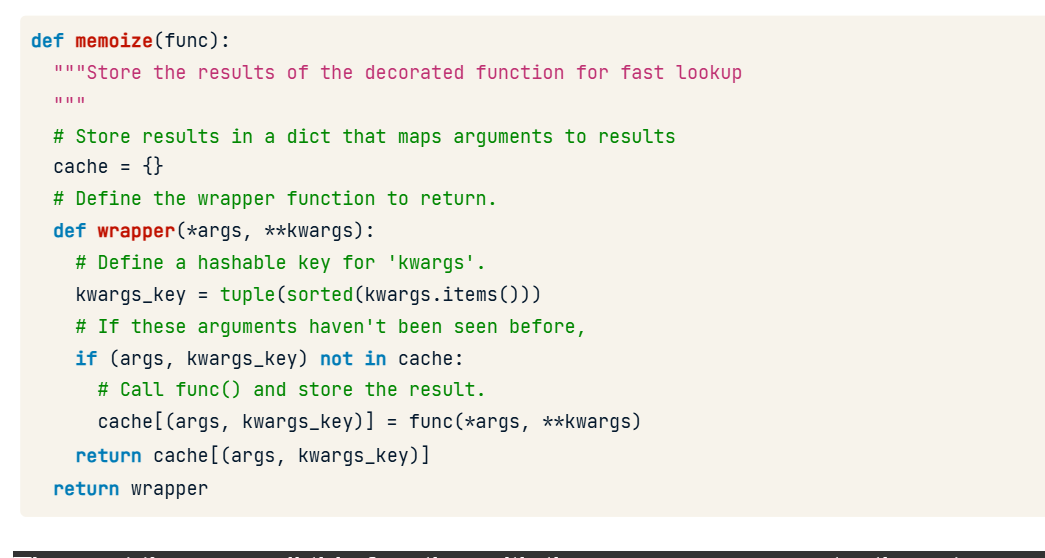
So if we decorate this simple sleep\_n\_seconds() function, you can see that sleeping for 5 seconds takes about 5 seconds, and sleeping for 10 seconds takes about 10 seconds. This is a trivial use of the decorator to show it working, but it can be very useful for finding the slow parts of your code.



**5. Memoizing**

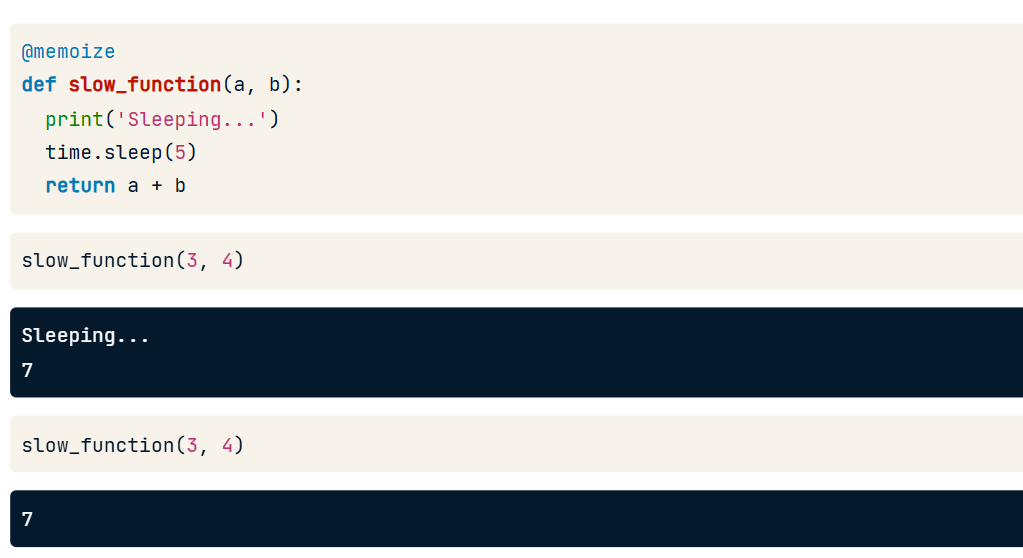
01:40 - 02:23

Memoizing is the process of storing the results of a function so that the next time the function is called with the same arguments; you can just look up the answer. We start by setting up a dictionary that will map arguments to results. Then, as usual, we create wrapper() to be the new decorated function that this decorator returns. When the new function gets called, we check to see whether we've ever seen these arguments before. If we haven't, we send them to the decorated function and store the result in the "cache" dictionary. Now we can look up the return value quickly in a dictionary of results. The next time we call this function with those same arguments, the return value will already be in the dictionary.



**Using memoize()**

Here we are memoizing slow\_function(). slow\_function() simply returns the sum of its arguments. In order to simulate a slow function, we have it sleep for 5 seconds before returning. If we call slow\_function() with the arguments 3 and 4, it will sleep for 5 seconds and then return 7. But if we call slow\_function() with the arguments 3 and 4 again, it will immediately return 7. Because we've stored the answer in the cache, the decorated function doesn't even have to call the original slow\_function() function.



**When to use decorators**

So when is it appropriate to use a decorator? You should consider using a decorator when you want to add some common bit of code to multiple functions. We could have added timing code in the body of all three of these functions, but that would violate the principle of Don't Repeat Yourself. Adding a decorator is a better choice.



**Decorators and metadata**

One of the problems with decorators is that they obscure the decorated function's metadata. In this lesson, I'll show you why it's a problem and how to fix it.

**2. Function with a docstring**

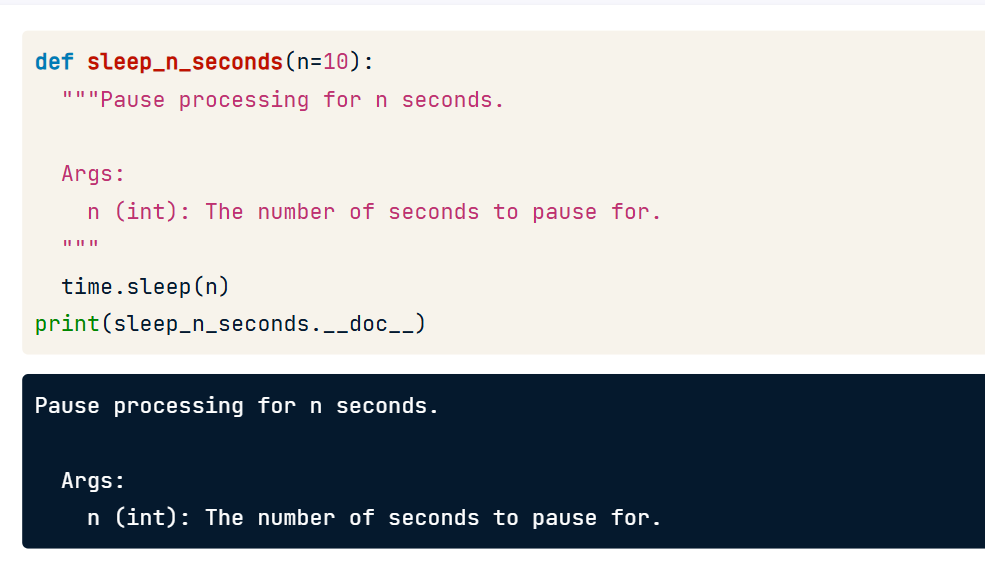
00:09 - 00:22

Here we have a nice function, sleep\_n\_seconds(), with a docstring that explains exactly what it does. If we look at the docstring attribute, we can see the text of the docstring.

**3. Other metadata**

00:22 - 00:29

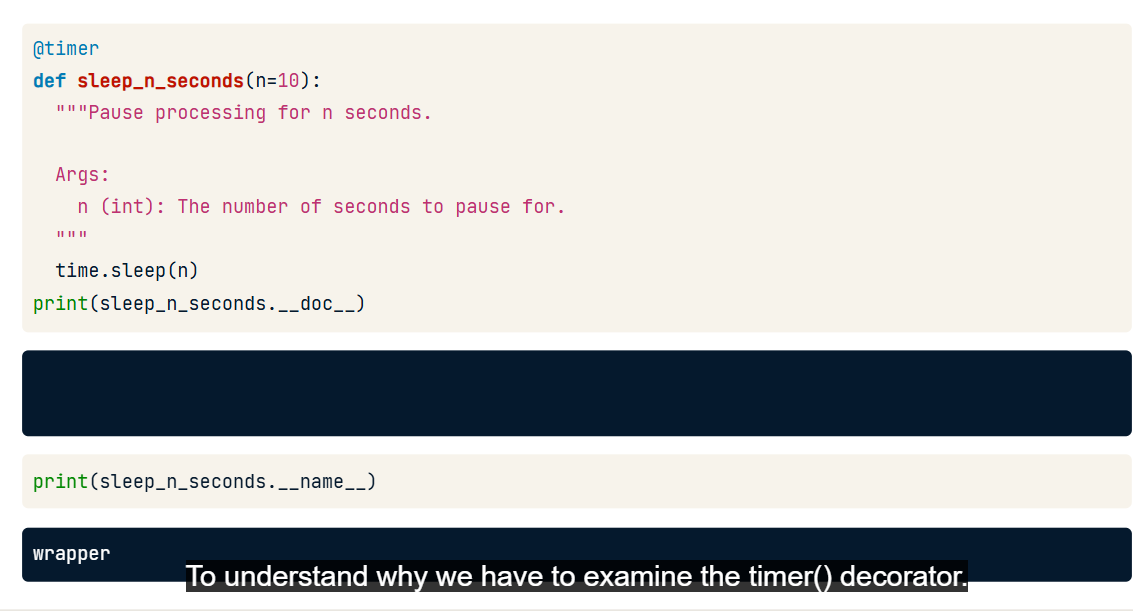
We can also access other metadata for the function, like its name and default arguments.



**4. A decorated function**

00:29 - 00:46

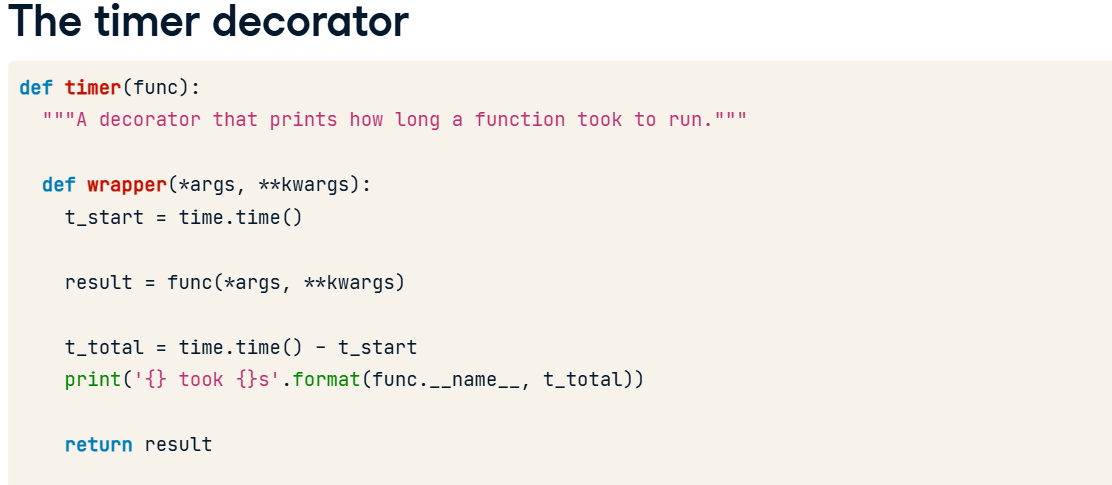
But watch what happens when we decorate sleep\_n\_seconds() with the timer() decorator as we've done here. When we try to print the docstring, we get nothing back. Even stranger, when we try to look up the function's name, Python tells us that sleep\_n\_seconds()'s name is "wrapper".



**5. The timer decorator**

00:46 - 01:13

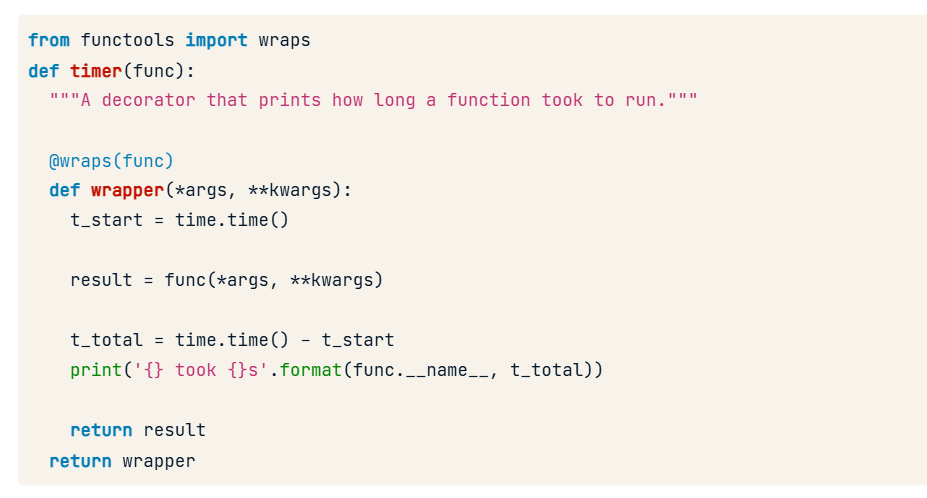
To understand why we have to examine the timer() decorator. Remember that when we write decorators, we almost always define a nested function to return. Because the decorator overwrites the sleep\_n\_seconds() function, when you ask for sleep\_n\_seconds()'s docstring or name, you are actually referencing the nested function that was returned by the decorator. In this case, the nested function was called wrapper() and it didn't have a docstring.



**6. functools.wraps()**

01:13 - 01:45

Fortunately, Python provides us with an easy way to fix this. The wraps() function from the functools module is a decorator that you use when defining a decorator. If you use it to decorate the wrapper function that your decorator returns, it will modify wrapper()'s metadata to look like the function you are decorating. Notice that the wraps() decorator takes the function you are decorating as an argument. We haven't talked about decorators that take arguments yet, but we will cover that in the next lesson.



**7. The metadata we want**

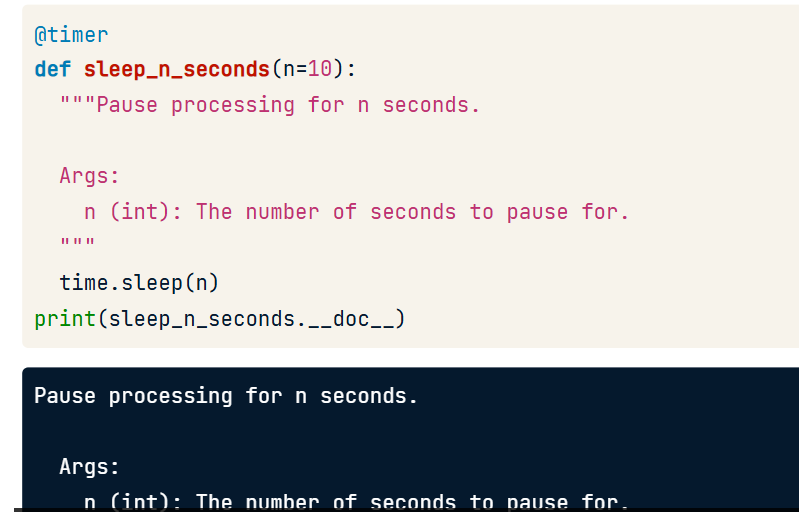
01:45 - 01:56

If we use this updated version of the timer() decorator to decorate sleep\_n\_seconds() and then try to print sleep\_n\_seconds()'s docstring, we get the result we expect.

**8. The metadata we want**

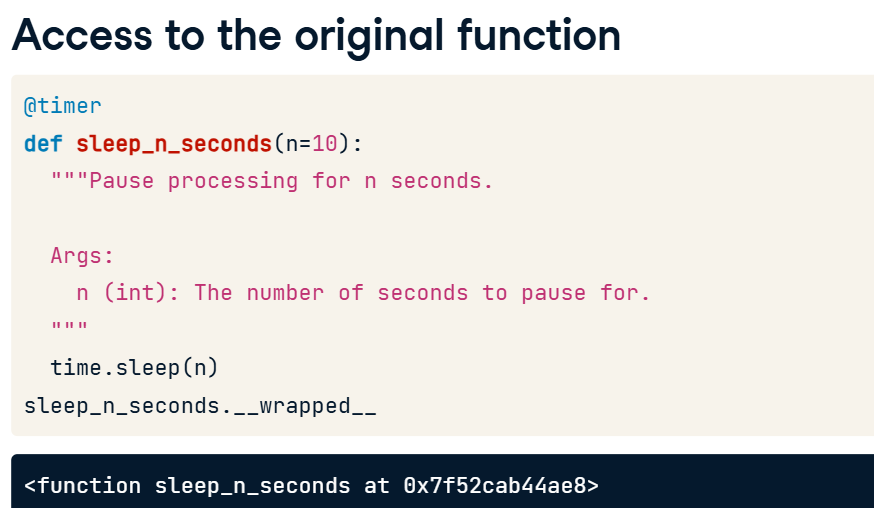
01:56 - 02:06

Likewise, printing the name or any other metadata now gives you the metadata from the function being decorated rather than the metadata of the wrapper() function.

**9. Access to the original function**

02:06 - 02:26

As an added bonus, using wraps() when creating your decorator also gives you easy access to the original undecorated function via the \_\_wrapped\_\_ attribute. Of course, you always had access to this function via the closure, but this is an easy way to get to it if you need it.



**Decorators that take arguments**

Sometimes it would be nice to add arguments to our decorators. To do that, we need another level of function nesting.

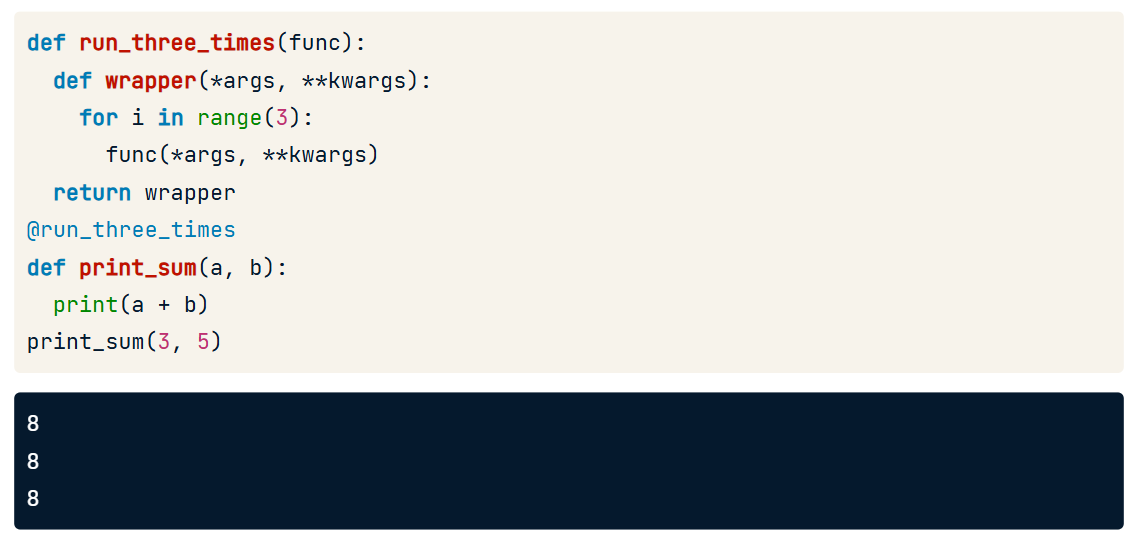
**2. run\_three\_times()**

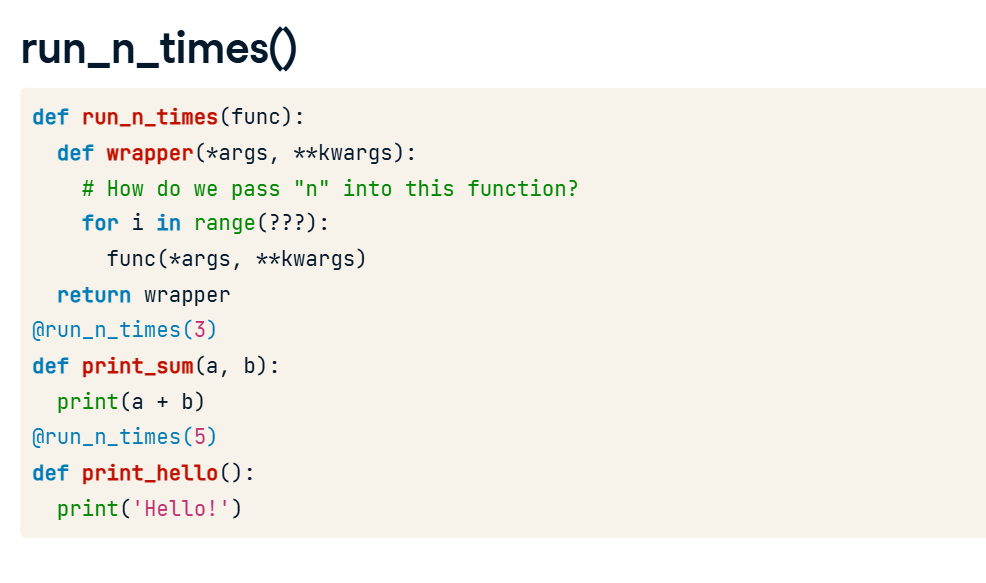
00:07 - 00:25

Let's consider this silly run\_three\_times() decorator. If you use it to decorate a function, it will run that function three times. If we use it to decorate the print\_sum() function and then run print\_sum(3,5), it will print 8 three times.

**3. run\_n\_times()**

00:25 - 00:59

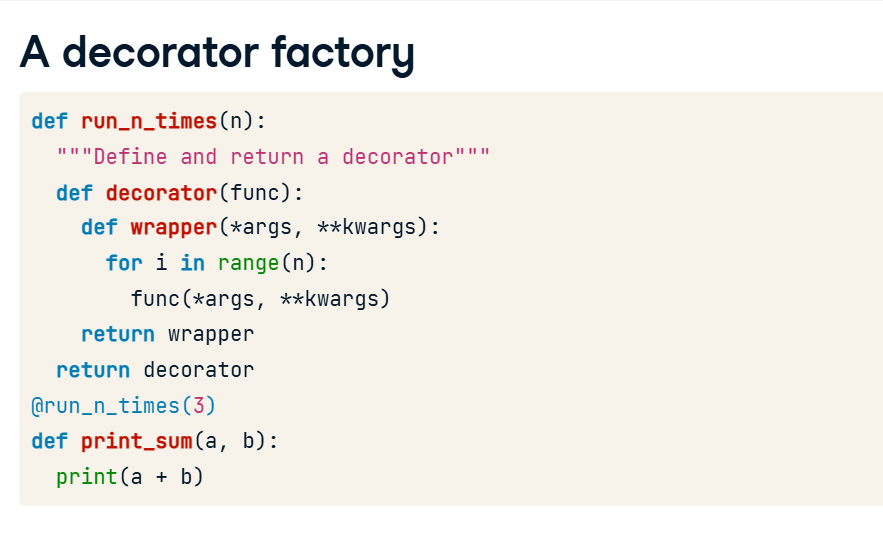
Let's think about what we would need to change if we wanted to write a run\_n\_times() decorator. We want to pass "n" as an argument, instead of hard-coding it in the decorator. If we had some way to pass n into the decorator, we could decorate print\_sum() so that it gets run three times and decorate print\_hello() to run five times. But a decorator is only supposed to take one argument - the function it is decorating. Also, when you use decorator syntax, you're not supposed to use the parentheses. So what gives?



**4. A decorator factory**

00:59 - 02:23

To make run\_n\_times() work, we have to turn it into a function that returns a decorator, rather than a function that is a decorator. So let's start by redefining run\_n\_times() so that it takes n as an argument, instead of func. Then, inside of run\_n\_times(), we'll define a new decorator function. This function takes "func" as an argument because it is the function that will be acting as our decorator. We start our new decorator with a nested wrapper() function, as usual. Now, since we are still inside the run\_n\_times() function, we have access to the n parameter that was passed to run\_n\_times(). We can use that to control how many times we repeat the loop that calls our decorated function. As usual for any decorator, we return the new wrapper() function. And, if run\_n\_times() returns the decorator() function we just defined, then we can use that return value as a decorator. Notice how when we decorate print\_sum() with run\_n\_times(), we use parentheses after @run\_n\_times. This indicates that we are actually calling run\_n\_times() and decorating print\_sum() with the result of that function call. Since the return value from run\_n\_times() is a decorator function, we can use it to decorate print\_sum().



**Expanded code**

This is a little bit confusing, so let me show you how this works without using decorator syntax. Like before, we have a function, run\_n\_times() that returns a decorator function when you call it. If we call run\_n\_times() with the argument 3, it will return a decorator. In fact, it returns the decorator that we defined at the beginning of this lesson, run\_three\_times(). We could decorate print\_sum() with this new decorator using decorator syntax. Python makes it convenient to do both of those in a single step though. When we use decorator syntax, the thing that comes after the @ symbol must be a reference to a decorator function. We can use the name of a specific decorator, or we can call a function that returns a decorator.



**Using run\_n\_times()**

To prove to you that it works the way we expect here is print\_sum() decorated with run\_n\_times(3). When we call print\_sum() with the arguments 3 and 5, it prints 8 three times. And we can just as easily decorate print\_hello(), which prints a hello message, with run\_n\_times(5). When we call print\_hello(), we get five hello messages, as expected.



**Timeout(): a real world example**

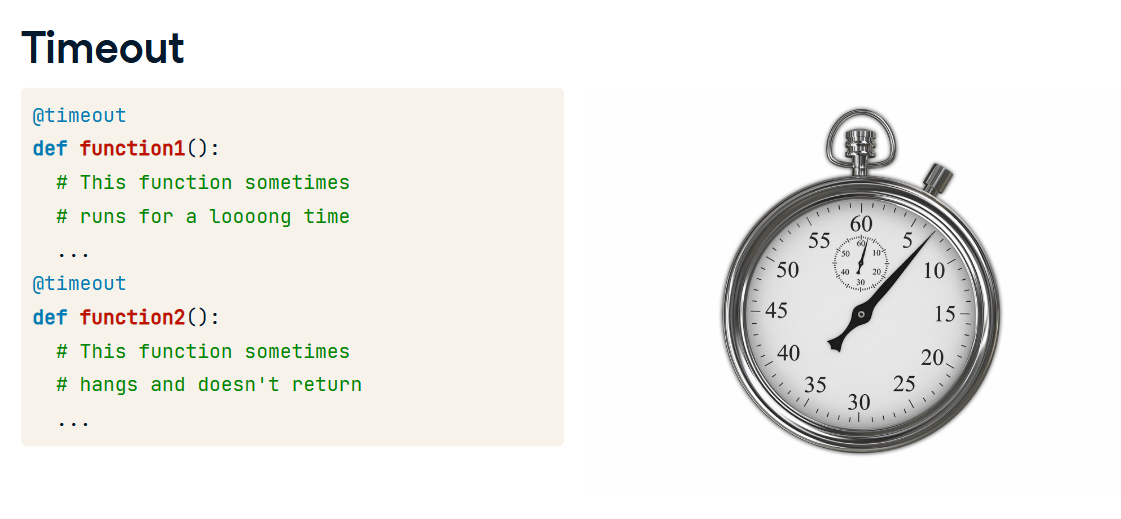
We're going to finish up by looking at an example of a real-world decorator that takes an argument to give you a better sense of how they work.

**2. Timeout**

00:07 - 00:15

For our first example, let's imagine that we have some functions that occasionally either run for longer than we want them to or just hang and never return.

It would be nice if we could add some kind of timeout() decorator to those functions that will raise an error if the function runs for longer than expected.



**4. Timeout - background info**

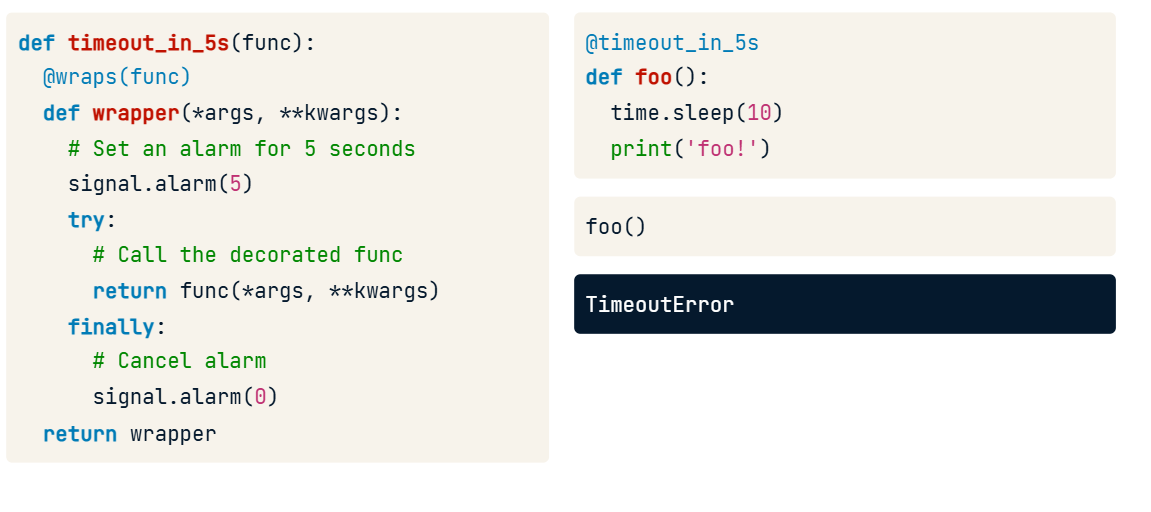
To create the timeout() decorator, we are going to use some functions from Python's signal module. These functions have nothing to do with decorators, but understanding them will help you understand the timeout() decorator I am about to show you. The raise\_timeout() function simply raises a TimeoutError when it is called. The signal() function tells Python, "When you see the signal whose number is signalnum, call the handler function." In this case, we tell Python to call raise\_timeout() whenever it sees the alarm signal. The alarm() function lets us set an alarm for some number of seconds in the future. Passing 0 to the alarm() function cancels the alarm.



**5. Timeout in 5 seconds**

01:07 - 02:09

We'll start by creating a decorator that times out in exactly 5 seconds, and then build from there to create a decorator that takes the timeout length as an argument. Our timeout\_in\_5s() decorator starts off by defining a wrapper() function to return as the new decorated function. Returning this function is what makes timeout\_in\_5s() a decorator. First wrapper() sets an alarm for 5 seconds in the future. Then it calls the function being decorated. It wraps that call in a try block so that in a finally block we can cancel the alarm. This ensures that the alarm either rings or gets canceled. Remember, when the alarm rings, Python calls the raise\_timeout() function. Let's use timeout\_in\_5s() to decorate a function that will definitely timeout. foo() sleeps for 10 seconds and then prints "foo!". If we call foo(), the 5-second alarm will ring before it finishes sleeping, and Python will raise a TimeoutErrror.



**6. Timeout in n seconds**

02:09 - 03:31

Now let's create a more useful version of the timeout() decorator. This decorator takes an argument. To decorate foo() we'll set the timeout to 5 seconds like we did previously. But when decorating bar(), we can set the timeout to 20 seconds. This allows us to set a timeout that is appropriate for each function. timeout() is a function that returns a decorator. I like to think of it as a decorator factory. When you call timeout(), it cranks out a brand new decorator that times out in 5 seconds, or 20 seconds, or whatever value we pass as n\_seconds. The first thing we need to do is define this new decorator that it will return. That decorator begins, like all of our decorators, by defining a wrapper() function to return. Now because n\_seconds is available to the wrapper() function we can set an alarm for n\_seconds in the future. The rest of the wrapper() function looks exactly like the wrapper() function from the timeout\_in\_5s() function. Notice that wrapper() returns the result of calling func(), decorator() returns wrapper, and timeout() returns decorator. So when we call foo(), which has a 5-second timeout, it will timeout like before. But bar(), which has a 20-second timeout, prints its message in 10 seconds, so the alarm gets canceled.

